ja.wikibooks/Haskell/圏論

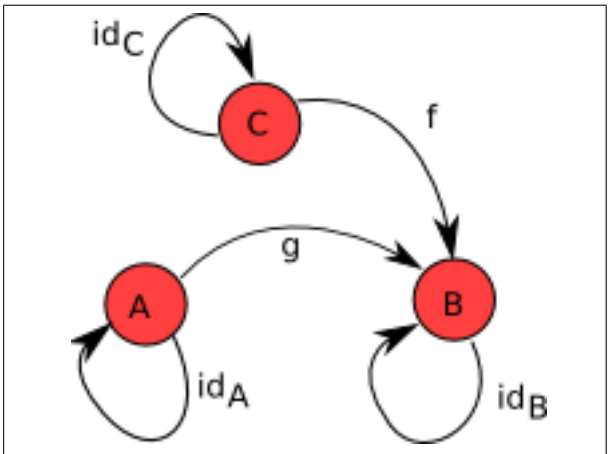
2012 年 2 月 2 日

この項目では Haskell に関連する内容に限って圏論の概観を与えることを試みる。そのために、数学的な定義に併せて Haskell コードも示す。絶対的な厳密さは求めない。そのかわり、圏論の概念とはどんなものか、どのように Haskell に関連するかの直観的な理解を読者に与えることを追求する。

1 圏の導入

本質的に、圏とは単純な集まりである。これは次の 3 つの要素からなる。

- **対象** (Object) の集まり。
- ふたつの対象 (**ソースオブジェクト** (source object) と **ターゲットオブジェクト** (target object)) をひとつに結びつける**射**の集まり。(これらは arrow と呼ばれることもあるが、Haskell ではこれは別の意味を持つ用語なので、ここではこの用語を避けることにする。) f がソースオブジェクト A からターゲットオブジェクト B への射であるとき、これを $f : A \to B$ と書く。
- これらの射の**合成**の概念。h が射 f, g の合成であるとき、これを $h = f \circ g$ と書く。



3 つの対象 A, B, C、3 つの恒等射 id_A, id_B, id_C と、さらに別の射 $f : C \to B, g : A \to B$ からなる単純な圏。3 つめの要素 (どのように射を合成するかの定義) は示していない。

様々なものが圏をつくる。例えば、**Set** はすべての集合を対象とし、射として標準的な関数、合成として通常の関数合成を持つ圏である。(圏の名前はしばしば太字で表される。) **Grp** はすべての群を対象とし、射として群の演算を保存する関数 (群準同型) を持つ圏である。つまり、任意の 2 つの群 G, H(それぞれ演算 $*, \cdot$ をもつ) について、$f(u * v) = f(u) \cdot f(v)$ ならば、かつその時に限り、$f : G \to H$ は **Grp** 内の射である。

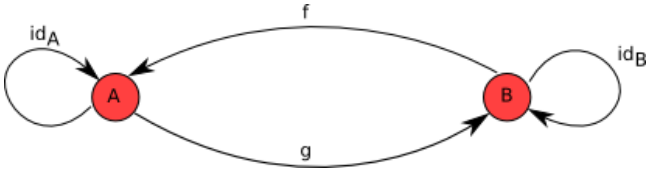
射はいつも関数のようにも思えるが、そうである必要はない。例えば、いかなる半順序 $(P, \leq)$ も圏を定義する。ここで対象は P の要素であり、任意の 2 つの対象 A, B について $A \leq B$ であるときかつその時に限り A と B の間に射が存在する。また、同じソースオブジェクトとターゲットオブジェクトを持つ複数の射が存在できる。**Set** の例でいえば、sin と cos はいずれも同じソースオブジェクト $\mathcal{R}$ とターゲットオブジェクト $[-1, 1]$ を持つが、もちろんこれらは同じ射ではない！

1.1 カテゴリ則

圏が従わなければならない規則が 3 つ存在する。最初に、非常に単純だが、射の合成は**結合的**である必要がある。つまり、

$$f \circ (g \circ h) = (f \circ g) \circ h$$

次に、圏は合成演算において**閉じて**いなければならない。すなわち、もし $f : B \to C$ かつ $g : A \to B$ ならば、$h = f \circ g$ であるような射 $h : A \to C$ がその圏に存在しなければならない。次の圏を使えば、これがどのように働くかを見ることができる。



f と g はどちらも射であるので、これらを合成したらこの圏にあるいずれかの射を得るはずである。ではどれが射 $f \circ g$ だろうか？唯一の答えは id_A である。同様に、$g \circ f = id_B$ もわかる。最後に、いかなる対象 A についても**恒等**射 $id_A : A \to A$、すなわちその射との合成が恒

等であるような射が圏 C に存在しなければならない。正確に言えば、任意の射 $g : A \to B$ に対して

$$g \circ id_A = id_B \circ g = g$$

である。

1.2 Haskell の圏 Hask

この記事で我々が考えようとしている中心の圏は、Haskell の型を対象とし、Haskell の関数を射とし、合成として $(\cdot)$ を使う圏 **Hask** である。型 A から B の関数 $f :: A \to B$ は **Hask** の射である。この定義が最初の 2 つの規則を満たすことは容易に確認できる。すなわち $(\cdot)$ は結合的であり、また任意の f と g について $f \cdot g$ は何らかの関数である。**Hask** における恒等射は id であり、次は自明である。[*1]

$$id \cdot f = f \cdot id = f$$

ただし、これは添え字を欠いてしまっており、上述の規則の厳密な翻訳ではない。Haskell の関数 id は**多相的**である。すなわちその定義域と値域に様々な異なる型をとることができる、もしくは圏で言い換えれば、異なるソース及びターゲットオブジェクトをとることができる。しかし圏論における射は**単相的**である。すなわちどんな射もソースオブジェクトとターゲットオブジェクトを 1 つずつ指定して定義される。多相な Haskell 関数はその型を明示することによって単相化することができる (単相型による**インスタンス化**) ため、『**Hask** 圏の型 A に対する恒等射は $(id :: A \to A)$ である』と書けばより正確になる。これを考慮すると、先程の則は次のように書き直せる。

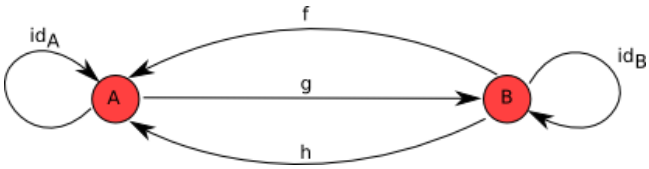
$$(id :: B \to B) \cdot f = f \cdot (id :: A \to A) = f$$

しかしながら議論を単純にするため、意味が明らかであるときはこの違いを無視することにする。

演習 1　さきほど触れたとおり、任意の半順序 $(P, \leq)$ は P の要素を対象、要素 a と b について $a \leq b$ であるときかつその時に限りその間に射をもつ圏である。上記の規則のどれが $\leq$ の推移性を保証しているか？

[*1] 原注: 実はここには微妙な部分が存在する。$(\cdot)$ は遅延評価関数であるため、もし f が *undefined* であると $id \cdot f = \lambda\_ \to \bot$ となる。さて、全ての意図と目的においてこれは $\bot\ (= f)$ と等しいように見えるかもしれないが、実は正格化関数 *seq* を使うことでこれらを区別することができる。これは圏の最後の規則が破られるということを意味する。(JOE:これはどれのこと？) 新たに正格な合成関数 $f \cdot! g = ((\cdot)\ \$!\ f)\ \$!\ g$ を定義すれば **Hask** を正しく圏にすることもできる。しかしここでは普通の $(\cdot)$ を使って進めていくことにする。全ての矛盾は *seq* が良い言語の性質をひどく破壊するということのせいにしておく。

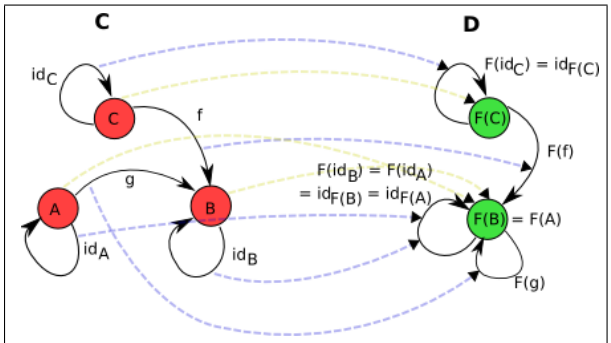
演習 2　(難しい) 上記の例に別の射を付け足すとこれは圏ではなくなる。なぜか？ヒント：合成演算の結合性を考えてみよ。



2 関手

対象と対象を関連付ける射を持つ幾つかの圏を見てきた。圏論の次の大きな話題は圏どうしを関連付ける**関手** (functor、ファンクタ) である。関手の本質は圏の間の変換である。つまり圏 C と D が与えられたとき、関手 $F : C \to D$ は次のことをする。

- C 内のあらゆる対象 A を、D 内の $F(A)$ に対応付ける。
- C 内の射 $f : A \to B$ を、D 内の $F(f) : F(A) \to F(B)$ に対応付ける。



2 つの圏 **C** と **D** の間の関手。対象の対応は黄色の、射の対応は青色の破線矢印で示した。注目すべきは対象 A と対象 B の両方が **D** の同じ対象に対応付けられていることである。それゆえ g はソースとターゲットが同じ対象である射に変換され (これは恒等射である必要はない)、id_A と id_B は同じ射となる。

関手の好例のひとつは忘却関手 (forgetful functor) **Grp** $\to$ **Set** である。これは群をその内在する集合に対応付け、群の射を同様に振る舞うがしかし群の代わりに集合の上で定義される関数に対応付ける。他の例としては冪集合関手 **Set** $\to$ **Set** がある。これは集合をその冪集合に対応付け、関数 $f : X \to Y$ を関数 $\mathcal{P}(X) \to \mathcal{P}(Y)$、つまり $U \subset X$ をとり $f(U) = \{ f(u) \mid u \in U \}$ と定義される、f の下の U の像を返す関数に対応付ける。あらゆる圏 C について、C 上の恒等関手として知られる関手、すなわち

対象や射をそれ自身に対応付ける、$1_C : C \to C$ を定義できる。これは後の 4 節で便利であることがじきにわかるであろう。

関手が従わなければならない幾つかの公理を挙げておこう。最初に、任意の対象 A 上の恒等射 id_A について、$F(id_A)$ は恒等射でなければならない。

$$F(id_A) \;=\; id_{F(A)}$$

2 つめに、関手は射の合成を分配しなければならない。

$$F(f \circ g) \;=\; F(f) \circ F(g)$$

演習 3　関手の図において、関手の規則が成り立ってることを確かめよ。

2.1　Hask における関手

Haskell でお目にかかったことがあるかもしれない型クラス *Functor* は、実際に圏論における関手の概念と一致する。関手は 2 つの構成要素からなることを思い出そう。関手はある圏の対象を別の圏の対象に対応付け、最初の圏の射をふたつめの圏の射に対応付ける。Haskell の関手は **Hask** から *func* である。ここで *func* はその関手の型の上でちょうど定義される **Hask** の部分圏である。例えばリスト関手は **Hask** から **Lst** である。ここで **Lst** は**リスト型**、すなわち任意の型 T に対するリスト型 $[T]$ のみを含む圏である。**Lst** における射はリスト型の上で定義される関数であり、これは任意の型 T, U に関して $[T] \to [U]$ の関数である。どのようにこれを Haskell の *Functor* 型クラスへと結びつけたらよいだろうか？定義を思い出そう。

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> (f a -> f b)
```

インスタンスの例も示しておこう。

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  = Nothing
```

ここが鍵となる部分である。**型構成子** *Maybe* は任意の型 T をとり新たな型 *Maybe* T になる。また、*Maybe* 型へと狭められた *fmap* は関数 $a \to b$ をとり関数 *Maybe* $a \to$ *Maybe* b を返す。まさにこれがそうではないか！ふたつの構成要素を定義した。ひとつは **Hask** の対象をとり別の圏 (これは *Maybe* 型を対象、*Maybe* 型上の関数を射とする) の対象を返す何か、もうひとつは **Hask** の射をとりこの圏の射を返す何か。ゆえに *Maybe* は関手である。

Haskell の関手が内容に対して map できるような型を表すということは有用な直観である。これはリストや *Maybe* だけでなく、木のようなより複雑な構造もそうである。構造の内容に map を行う関数は *fmap* を使って書くことができ、どんな関手構造もこの関数に渡すことができる。例えば、*Data.List.map*, *Data.Map.map*, *Data.Array.IArray. amap* などを全てカバーするジェネリックな関数を書くことができる。

関手の公理はどうだろうか？任意の A に対する id_A として多相的な関数 *id* を使い、最初の則は次のように書ける。

$$fmap\ id \;=\; id$$

先程の直観によると、この式は「構造を辿ってしかし各要素に何もしない map と、全く何もしないことは等しい」と解釈できる。次に、射の合成は単に $(\cdot)$ である。つまり、

$$fmap\ (f \cdot g) \;=\; fmap\ f \cdot fmap\ g$$

この 2 つめの則は実用的に極めて役に立つ。関手をリストか何かのコンテナと考えると、右辺は 2 パスのアルゴリズムを表す。つまり構造についてまず g を map し、さらに f を map する。この関手の公理はこのような 2 パスのアルゴリズムを $f \cdot g$ を行う 1 パスのアルゴリズムへと変換できることを保証する。この処理は**融合** (fusion) として知られている。

演習 4　リスト関手と *Maybe* 関手について、これらの規則を確かめよ。

2.2　圏論の概念から Haskell への翻訳

関手はどのように圏論が Haskell へと翻訳されるかの良い例を提供する。次が理解の鍵となる。

- 圏 **Hask** とその部分圏を扱う。
- 対象は型である。
- 射は関数である。
- 型を取り別の型を返すものが型構成子である。
- 関数をとり別の関数を返すものが高階関数である。
- 圏論ではしばしば一度に多くの対象にわたって物事が定義されるという事実を、型クラスとそれが同時に提供する多相性はうまく捉えている。

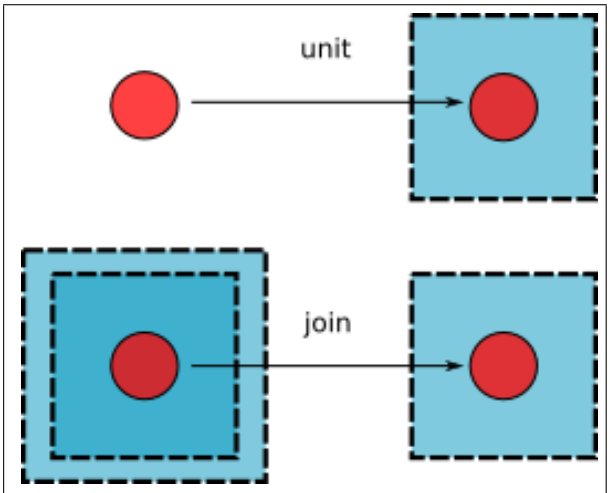
3　モナド

モナドは言うまでもなく Haskell の極めて重要な概念であるが、実はこれらは本来圏論に由来するものである。**モナド**は圏から同じ圏の関手であり、さらにいくつかの追

加の構造を提供する。では定義を見てみよう。モナドは C 内のすべての対象 X に対して 2 つの射[*2] を伴う関手 $M : C \to C$ である。

$$unit^M_X : X \to M(X)$$
$$join^M_X : M(M(X)) \to M(X)$$

議論されているモナドが明らかなとき、これらの関数から上添字 M を省略することにし、単に何らかの X にする $unit_X$ や $join_X$ について述べる。



モナドの全ての対象について *unit* と *join* の 2 の射が存在しなくてはならない。

では、どのようにこれが Haskell の型クラス *Monad* に翻訳されるかを見よう。

```
class Functor m ⇒ Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

このクラス制約 *Functor m* は m があらかじめ関手の構造、つまり対象と射の対応付けを持っていることを保証する。*return* は全ての X に対する $unit_X$ の (多相的な) 類義語である。しかしここには問題がある。*return* の型は *unit* のそれに非常に似ているけれども、もう一方の関数、しばしば *bind* と呼ばれる $>>=$ の型は *join* に似ていない。ここでさらに別のモナド関数 $join :: Monad\ m \Rightarrow m\ (m\ a) \to m\ a$ があり、これの型は非常によく似ている。実際に、*join* と $(>>=)$ は次のように互いに補い合うことができる。

```
join :: Monad m ⇒ m (m a) → m a
join x = x >>= id
(>>=) :: Monad m ⇒ m a → (a → m b) → m b
x >>= f = join (fmap f x)
```

つまり、モナドの *return*, *fmap*, *join* を定義することは、*return* と $(>>=)$ を定義することと同等である。圏論でモナドを定義する典型的な方法が *unit* と *join* を与えることであるのに対して、Haskell プログラマは *return* と $(>>=)$ を与えることを好むことがわかる[*3] 多くの場合、圏論の方法のほうが理にかなっている。何らかの構造 M があり、任意の対象 X を $M(X)$ としたり、$M(M(X))$ を $M(X)$ とできる自然な方法があるとき、それはしばしばモナドである。次の節で例を示す。

3.1 例: 冪集合関手もモナドである

先ほど述べた冪集合関手 P : **Set** $\to$ **Set** はモナドをなす。任意の集合 S について、要素をその単集合へ写す $unit_S(x) = \{x\}$ を考える。これらの単集合のそれぞれは明らかに S の部分集合であり、よって $unit_S$ は S の冪集合の要素を返す、つまりモナドの要求を満たしている。同様に関数 $join_S$ を次にように定義できる。引数として $L \in \mathcal{P}(\mathcal{P}(S))$ をとる。これは、

- 冪集合の要素は元の集合の部分集合である、すなわち $X \in \mathcal{P}(S) \leftrightarrow X \subseteq S$
- つまり、$L \subseteq \mathcal{P}(S)$
- つまり、L は S の部分集合の集合。

よって $joins_S$ はこれらの部分集合の和を返すことで、S のまた別の部分集合を与えるようにする。式で表せば、

$$join_S(L) = \bigcup L$$

従って、(次の節で探求する規則を満たすことが証明できれば)P はモナドである。

実際のところ、議論の対象が集合の代わりにリストになるという違いを除けば、P はほとんどリストモナドと等しい。これらはほぼ同じものである。表 1 に対比を示す。

[*2] 原注: 熟達した圏論の理論家は、ここで少し単純化していることに気付くだろう。*unit* と *join* を自然変換として導入する代わりに、これらを明示的に射として扱った。そして naturality を 4.3 節の第 3,4 規則として追加した。ここでは圏論全体を教えようとしているわけではなく、Haskell のある構造に対する理論的裏付けとしての圏論だけを示したいために、このような単純化を行った。これらの射に Haskell で対応するものを連想させる名前を付けていることに気付く読者もいるかも知れない。これは名前 η と μ が直観的でないためである。(JOE:オリジナルがリンク切れてる。)

[*3] 原注: これはおそらく、圏論では様々な構造のコンテナとしての側面に重点が置かれるのに対して、Haskell プログラマはモナドを共通する機能を持つ計算を連鎖させる方法と考えることを好むことが原因であろう。*join* はコンテナのふたつの層をひとつに潰すという意味でコンテナと自然に関連し、対して $(>>=)$ は何かを行い、その結果を何かに与えるという意味で計算を連鎖させる自然な演算子である。

Set に対する冪集合関手		Haskell のリストモナド	
関数の型	**定義**	**関数の型**	**定義**
集合を S、射を $f : A \to B$ とする		型を T、関数を $f :: A \to B$ とする	
$P(f) : \mathcal{P}(A) \to \mathcal{P}(B)$	$(P(f))(S) = \{f(a) \mid a \in S\}$	$fmap\ f\ ::\ [A]\ \to\ [B]$	$fmap\ f\ xs\ =\ [\,f\ a \mid a\ \leftarrow\ xs\,]$
$unit_S\ :\ S\ \to\ \mathcal{P}(S)$	$unit_S(x)\ =\ \{\,x\,\}$	$return\ ::\ T\ \to\ [T]$	$return\ x\ =\ [x]$
$join_S\ :\ \mathcal{P}(\mathcal{P}(S))\ \to\ \mathcal{P}(S)$	$join_S(L)\ =\ \bigcup L$	$join\ ::\ [\,[T]\,]\ \to\ [T]$	$join\ xs\ =\ concat\ xs$

表 1　冪集合関手とリストモナドの比較

4　モナド則とその重要性

関手がそう呼ばれるためには従わなければならない公理があるように、モナドにも幾つかそのような公理が存在する。まずはそれらを一覧し、次に Haskell へと翻訳し、なぜそれが重要なのかをみていくことにしよう。

モナド $M : C \to C$ と、対象 $A, B \in C$ に射 $f : A \to B$ があるとする。次が成り立たなければならない。

1. $join \circ M(join)\ =\ join \circ join$
2. $join \circ M(unit)\ =\ join \circ unit\ =\ id$
3. $unit \circ f\ =\ M(f) \circ unit$
4. $join \circ M(M(f))\ =\ M(f) \circ join$

Haskell への翻訳はなるべく自明であってほしい。

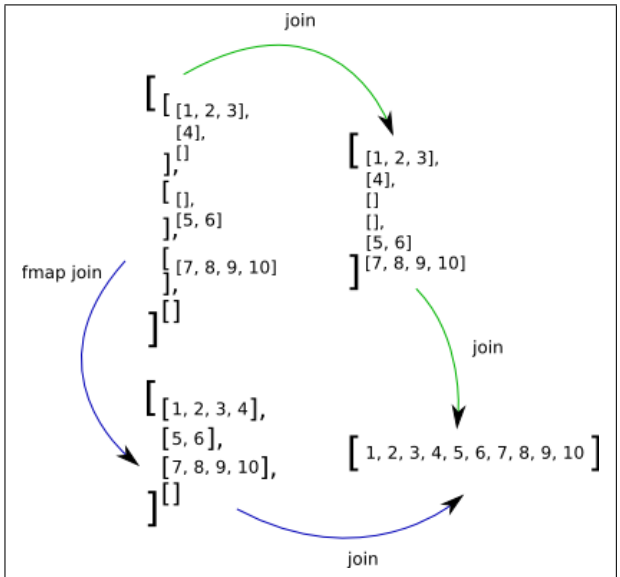
1. $join\ \cdot\ fmap\ join\ =\ join\ \cdot\ join$
2. $join\ \cdot\ fmap\ return\ =\ join\ \cdot\ return\ =\ id$
3. $return\ \cdot\ f\ =\ fmap\ f\ \cdot\ return$
4. $join\ \cdot\ fmap\ (fmap\ f)\ =\ fmap\ f\ \cdot\ join$

(*fmap* が関手の構成要素の一つで、射に作用するものであることを思い出そう。) これらの則は最初は少し不可解にも思える。いったいこれらの則はどういう意味で、なぜモナドはこれを満たすべきなのだろうか？ではこれらの法則について調べていこう。

4.1　最初の規則

$$join\ \cdot\ fmap\ join\ =\ join\ \cdot\ join$$

この規則を理解するために、まずはリストの例を使っていこう。最初の規則は $join\ \cdot\ fmap\ join$ (左辺) と $join\ \cdot\ join$ (右辺) の 2 つの関数について述べている。これらの型はどうなるだろうか？ (今はリストモナドについて議論しているので) *join* の型は $[[a]]\ \to\ [a]$ であることを思い出そう。つまり両辺の型はどちらも $[[[a]]]\ \to\ [a]$ である。(これが同じであるという事実は役に立つ。最終的には、両辺が全く同じ関数であることを示そうとしているのだ。) では、リストのリストのリストがあるとしよう。



最初の規則のリスト版の説明。リストモナドでは *join* は *concat* で、*fmap* は *map* であることを思い出そう。

左辺はこの三重のリストに $fmap\ join$ を作用させ (1)、その結果に *join* を適用する (2)。*fmap* はリストに対しては *map* と同じであるので、(1) の計算はまずいちばん外側のリストの要素に対して *join* を map する。各要素に対して、*join* はリストのリストそれぞれを連結する。結果としてリストのリストが得られたら、(2) によりこれ全体に *join* を適用する。手短に言えば、トップレベルに**入り込み**、二番目と三番目のレベルを潰して 1 つにして、次いでトップレベルとこの新しいレベルを潰して 1 つにする。

では、右辺についてはどうだろうか？最初にリストのリストのリストに対して *join* を実行する。普通 *join* は二重のリストに適用するものであるが、三重のリストにも適用できる。なぜなら、$[[[a]]]$ は $b\ =\ [a]$ とおけば $[[b]]$ と書けるからである。つまり三重のリストはある意味で二重のリストにすぎない、ただし最深部は**フラット**な値でなく別のリストからなっている。だからリストのリスト (のリスト) に *join* を適用すると、外側の 2 つのリストが 1 つに平坦化される。外から 2 番目の深さのリストの要素はフラット

ではなく 3 番目の層のリストが含まれているので、この結果はリストのリストであり、次にもう一方の *join* がこの構造を平坦化する。要約すると、左辺は内側のふたつの層を新しい層へと平坦化し、それから最も外側と新しい層を平坦化する。右辺は外側のふたつの層を平坦化し、それから新しい層と最も内側の層を平坦化する。これらの二つの操作は等しい。これはいわば *join* に関する結合則のようなものである。

次のようにした *Maybe* はモナドである。

$$
\begin{aligned}
&return :: a \to Maybe\ a\\
&return\ x = Just\ x\\
&join :: Maybe\ (Maybe\ a) \to Maybe\ a\\
&join\ Nothing = Nothing\\
&join\ (Just\ Nothing) = Nothing\\
&join\ (Just\ (Just\ x)) = Just\ x
\end{aligned}
$$

三重の *Maybe* (これは *Nothing*, *Just Nothing*, *Just* (*Just Nothing*), *Just* (*Just* (*Just* x)) といった値をとる) を考えて、ひとつめの規則は、内側のふたつの層を先に潰しそれから外側の層を潰すことは、外側の層を先に潰しそれから最も内側の層を潰すことと全く同じであると述べている。

演習 5　層の平坦化がどのように働くかを見るために、リストモナドと *Maybe* モナドがこの規則に確かに従うことを、幾つかの例について確認せよ。

4.2　二番目の規則

$$join \cdot fmap\ return = join \cdot return\ (= id)$$

それでは、二番目の規則はどうだろうか？再び、リストの例で始めよう。二番目の規則で述べられている両辺は $[a] \to [a]$ の関数である。左辺はまずリストのそれぞれの要素 x を要素がそれひとつだけのリスト $[x]$ へと変え、シングルトンリストのリストを得る。この二重のリストは再び *join* を使って一重のリストへと平坦化される。以上が左辺の表す関数である。一方右辺では、リスト全体 $[x, y, z, \cdots]$ をとり、これをリストのシングルトンリスト $[\,[x, y, z, \cdots]\,]$ にし、それから再びこのふたつの層をひとつに平坦化する。この規則は一言で説明できるほどの自明さはないが、*return* をモナドの値に適用しその結果を *join* することは、*return* を最上位の層の内側で実行しても外側で実行しても同じ結果になるべきであるということを本質的に述べている。

演習 6　*Maybe* モナドについて、二番目の規則を証明せよ。

4.3　3 番目と 4 番目の規則

$$
\begin{aligned}
&return \cdot f = fmap\ f \cdot return\\
&join \cdot fmap\ (fmap\ f) = fmap\ f \cdot join
\end{aligned}
$$

最後のふたつの法則は、どのようにモナドが振舞うべきかに関するもっと自明な事実を示す。

演習 7　最初と二番目の規則を説明したのと同様の方法で、それらがどのような意味を持つかを探り、これらの規則がどんなモナドについても成り立つべきであることを確かめよ。

4.4　do ブロックへの適用

モナドが従わなければならない規則について直観的に説明した。ではなぜこれは重要なのか？その答えは do ブロックについて考えたときに明らかになる。次のお決まりの変換が示しているように、do ブロックが文の組み合わせを ($\gg=$) を使わずに表すためのただの構文糖衣であることを思い出そう。

$$
\begin{aligned}
&\mathbf{do}\,\{\,e\,\} &&\longrightarrow e\\
&\mathbf{do}\,\{\,\mathbf{let}\,\{\,x = d\,\};\ e\,\} &&\longrightarrow \mathbf{let}\ x = d\ \mathbf{in}\ \mathbf{do}\,\{\,e\,\}\\
&\mathbf{do}\,\{\,x \leftarrow d;\ e\,\} &&\longrightarrow d \gg= \lambda x \to \mathbf{do}\,\{\,e\,\}\\
&\mathbf{do}\,\{\,d;\ e\,\} &&\longrightarrow d \gg= \lambda \_ \to \mathbf{do}\,\{\,e\,\}
\end{aligned}
$$

ここで用いている 4 つの規則から、通常の *return* と ($\gg=$) を使ったモナド則を証明できることに注意したい。(非常に大掛かりな証明もあるので、飛ばしても構わない。)

$return\ x \gg= f = f\ x$

証明:

$$
\begin{aligned}
&\quad return\ x \gg= f\\
&= join\ (fmap\ f\ (return\ x)) && \text{— } \gg= \text{の定義より}\\
&= join\ (return\ (f\ x)) && \text{— 規則 3 より}\\
&= (join \cdot return)\ (f\ x)\\
&= id\ (f\ x) && \text{— 規則 2 より}\\
&= f\ x
\end{aligned}
$$

$m \gg= return = m$

証明:

$$
\begin{aligned}
&\quad m \gg= return\\
&= join\ (fmap\ return\ m) && \text{— } \gg= \text{の定義より}\\
&= (join \cdot fmap\ return)\ m\\
&= id\ m && \text{— 規則 2 より}\\
&= m
\end{aligned}
$$

$(m \gg= f) \gg= g = m \gg= (\lambda x \to f\ x \gg= g)$

証明は図 1 に示す。($fmap\ f \cdot fmap\ g = fmap\ (f \cdot g)$ を用いる。)

$$
\begin{aligned}
&(m \gg\!= f) \gg\!= g \\
=&(join\ (fmap\ f\ m)) \gg\!= g && \text{— } (\gg\!=) \text{ の定義より} \\
=&join\ (fmap\ g\ (join\ (fmap\ f\ m))) && \text{— } (\gg\!=) \text{ の定義より} \\
=&(join \cdot fmap\ g)\ (join\ (fmap\ f\ m)) \\
=&(join \cdot fmap\ g \cdot join)\ (fmap\ f\ m) \\
=&(join \cdot join \cdot fmap\ (fmap\ g))\ (fmap\ f\ m) && \text{— 規則 4 より} \\
=&(join \cdot join \cdot fmap\ (fmap\ g) \cdot fmap\ f)\ m \\
=&(join \cdot join \cdot fmap\ (fmap\ g \cdot f))\ m && \text{— 関手の分配則より} \\
=&(join \cdot join \cdot fmap\ (\lambda x \to fmap\ g\ (f\ x)))\ m \\
=&(join \cdot fmap\ join \cdot fmap\ (\lambda x \to fmap\ g\ (f\ x)))\ m && \text{— 規則 1 より} \\
=&(join \cdot fmap\ (join \cdot (\lambda x \to fmap\ g\ (f\ x))))\ m && \text{— 関手の分配則より} \\
=&(join \cdot fmap\ (\lambda x \to join\ (fmap\ g\ (f\ x))))\ m \\
=&(join \cdot fmap\ (\lambda x \to f\ x \gg\!= g))\ m && \text{— } (\gg\!=) \text{ の定義より} \\
=&join\ (fmap\ (\lambda x \to f\ x \gg\!= g)\ m) \\
=&m \gg\!= (\lambda x \to f\ x \gg\!= g) && \text{— } (\gg\!=) \text{ の定義より}
\end{aligned}
$$

図 1　$\gg\!=$ の結合則の証明

ポイントフリー形式	**do ブロック形式**
$return\ e \gg\!= f\ =\ f\ e$	$\mathbf{do}\,\{\, x \leftarrow return\ e;\ f\ x\,\}\ =\ \mathbf{do}\,\{\, f\ e\,\}$
$m \gg\!= return\ =\ m$	$\mathbf{do}\,\{\, x \leftarrow m;\ return\ x\,\}\ =\ \mathbf{do}\,\{\, m\,\}$
$(m \gg\!= f) \gg\!= g\ =\ m \gg\!= (\lambda x \to f\ x \gg\!= g)$	$\mathbf{do}\,\{y \leftarrow \mathbf{do}\,\{\, x \leftarrow m;\ f\ x\,\};\ g\ y\,\}$ $=$ $\mathbf{do}\,\{x \leftarrow m;\ y \leftarrow f\ x;\ g\ y\,\}$

表 2　ポイントフリー形式と do ブロック形式の対応

return と $(\gg\!=)$ を使うこの新たなモナド則は、これを do 記法へと変換できる。その対応を表 2 に示す。

モナド則はいまや do ブロックがどのように機能すべきかという常識を述べた文である。もしこれらの規則が一つでも無効であれば、do ブロックを期待したとおりに操作できなくなるのでユーザが混乱する。要するに、モナド則はユーザビリティガイドラインなのである。

演習 8　規則をここでは 2 バージョン示した。

— 圏論的

$$
\begin{aligned}
&join \cdot fmap\ join\ =\ join \cdot join \\
&join \cdot fmap\ return\ =\ join \cdot return\ =\ id \\
&return \cdot f\ =\ fmap\ f \cdot return \\
&join \cdot fmap\ (fmap\ f)\ =\ fmap\ f \cdot join
\end{aligned}
$$

— 関数的

$$
\begin{aligned}
&m \gg\!= return\ =\ m \\
&return\ m \gg\!= f\ =\ f\ m \\
&(m \gg\!= f) \gg\!= g\ =\ m \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

これらは全く同等である。圏論的な規則から関数的な規則を作る方法は示した。逆をやってみよう。関数的な規則から始めて、圏論の規則が満たされることを示せ。以下の定義を思い出すと役に立つ。

$$
\begin{aligned}
&join\ m\ =\ m \gg\!= id \\
&fmap\ f\ m\ =\ m \gg\!= return \cdot f
\end{aligned}
$$

この問題を薦めてくれた Yitzchak Gale に感謝する。

5 まとめ

この章で長い道のりを辿ってきた。圏とは何か、どのように Haskell に応用されているのかを見てきた。関手をはじめとする圏論の基本的な概念から、モナドのようなより発展的な内容まで、それらがいかに Haskell らしさに必要不可欠であるかについても紹介してきた。自然変換 (natural transformations) など、我々の目的に必要なかった幾つかの圏論の基本的事項については触れなかったが、代わりに Haskell の構造の背後に根付く圏論の直観的な理解を提供した。

付録 A 演習の解答

■演習 1 合成演算に関して閉じているという規則により、P の要素 a, b, c について $a \leq b, b \leq c$ であるとき $a \leq c$ に対応する射も存在する。

■演習 2 図の圏で f と h が異なる射であると仮定する。先の議論から $f \circ g = id_B$ である。また同様の議論により $g \circ h = id_A$ となる。ここで合成の結合則を考える。恒等射の性質も用いて計算すると左辺は $f \circ (g \circ h) = f \circ id_A = f$ となり、右辺は $(f \circ g) \circ h = id_B \circ h = h$ となる。この二つが等しい $(f = h)$ ことは仮定と矛盾する。すなわち背理法よりこれは圏ではないことになる。

■演習 3 $\mathbf{C}$ の恒等射はいずれも $\mathbf{D}$ の恒等射に写っている。合成をどうするかの定義がないので、分配則については確認できない。

■演習 4 (方針) 規則の両辺はそれぞれ関数なので、全ての値に関して等しいことを示す。リストについては、リストの構造に関する帰納法で、$[\,]$ の場合と $(x : xs)$ の場合に分けて証明する。*Maybe* については、*Nothing* と *Just* x の場合分けで証明する。

■演習 5 (方針) リストについては図の例で充分であろう。*Maybe* については、本文に示されている 4 種類の構造全てについて計算を行なってみる必要がある。

■演習 6 (方針) 普通に *Nothing* と *Just* x の場合分けで証明する。

■演習 7 3 番目の法則の両辺は、$f :: a \to b$ とすると $a \to M(b)$ の関数であり、左辺では先に f を計算してからそれを *return* で構造に入れている。右辺では先に構造に入れてから、その中身に f をするために *fmap* を使っている。いずれの順で計算しても同じになるのが自然と考えられる。*fmap* と *return* の交換則に相当する。

4 番目の法則の両辺は、同じく $f :: a \to b$ として $M(M(a)) \to M(b)$ の関数であり、左辺は二重の構造の中にある値に f を二重の *fmap* により作用させておいて、後で *join* によって外側の構造を潰している。右辺では先に外側の構造を潰してしまってから、一重の構造の中に f を一重の *fmap* で作用させている。*fmap* と *join* の交換則に相当する。

■演習 8 難問だと書いてないけど大丈夫なのかな。(後でやる)